Comparaison de moteurs exécutifs pour la parallélisation de solveurs linéaires itératifs

Adrien Roussel

Compas'2016 : Parallélisme / Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Comparaison de moteurs exécutifs pour la parallélisation de solveurs linéaires itératifs

Adrien Roussel[1,2]

[1]Inria Rhône Alpes, Équipe Projet AVALON
Laboratoire LIP – ENS Lyon – 46 Allée d'Italie
69364 Lyon – France

[2]IFP Énergies Nouvelles,
Dpt d'Informatique Scientifique – 1 & 4 Avenue de Bois Préau
92500 Rueil-Malmaison – France
adrien.roussel@{inria,ifpen}.fr

Résumé
Dans le domaine de la simulation de réservoir, un grand système linéaire creux doit être résolu à chaque itération d'une méthode de Newton, ce qui est très consommateur en temps. Des méthodes numériques itératives sont utilisées pour résoudre ces équations. Elles rejouent la même séquence d'opérations jusqu'à converger vers la solution. Pour des systèmes non-structurés comme ceux que nous rencontrons, les méthodes itératives de Krylov comme le BiCGStab sont très utilisées. La parallélisation que nous proposons repose ici sur un modèle à base de tâches avec dépendances de données. Nous expérimentons notre implémentation au dessus de plusieurs moteurs exécutifs tels que : OpenMP, OmpSs, X-Kaapi et HARTS. Nous mettons en avant l'impact, parfois néfaste, que peut avoir la gestion des tâches. Un surcoût trop élevé peut freiner l'exécution, mais également un mauvais placement de tâche sur une architecture hiérarchique. La répétition à chaque itération de ces éléments peuvent impacter l'exécution.

Mots-clés : moteur exécutif, calcul parallèle, solveur linéaire, méthode itérative

---

1. Introduction

Dans le domaine de la simulation de réservoirs de pétrole, les modèles d'écoulements en milieux poreux conduisent à la discrétisation de systèmes d'équations aux dérivées partielles (EDP) selon un schéma aux volumes finis, qui amènent à la résolution d'un système non-linéaire avec une méthode itérative de Newton. À chaque itération, un système linéaire creux non structuré doit être résolu. La méthode du gradient biconjugué (BiCGStab) [15] est un solveur linéaire fréquemment utilisé pour ce type de systèmes.
Les moteurs exécutifs sont des outils qui permettent d'exploiter efficacement les architectures hétérogènes émergentes. Chacun d'eux possède ses propres mécanismes internes qui sont plus ou moins efficaces selon les propriétés algorithmiques de l'application. Beaucoup de moteurs exécutifs récents reposent sur un modèle de programmation par tâches. Ces outils se doivent de porter une attention particulière aux méthodes itératives, notamment aux surcoûts qui se cumulent au cours des itérations.

Nous menons ici une étude comparative de différents moteurs exécutifs reposant sur un modèle de tâches avec dépendances de données, afin d'évaluer leurs performances sur des méthodes itératives d'algèbre linéaire. Même si certains moteurs exécutifs permettent d'exécuter des applications sur des architectures à mémoire distribuée, dans cet article, nous nous restreignons à l'exploitation de processeurs multi-cœurs à mémoire partagée afin de montrer comment profiter pleinement des performances au sein d'un noeud de calcul sur machine homogène. Nous avons implémenté une méthode itérative à destination des architectures NUMA multi-cœurs avec différents moteurs exécutifs. Nous comparons chaque implémentation parallèle, afin de mettre en évidence les impacts que peuvent avoir ces outils sur les performances d'une méthode itérative. Nous définissons plusieurs critères de comparaison que nous appliquons pour aider à analyser les résultats d'expériences sur une architecture NUMA. À l'exécution, nous cherchons à évaluer aussi bien le surcoût par itération que le placement des tâches selon la topologie de la machine.
Dans un premier temps, nous détaillons chacun des moteurs exécutifs utilisés dans cette étude. Nous présentons ensuite un moyen de décrire de manière unique un algorithme itératif, indépendamment des moteurs exécutifs, grâce au développement d'une API d'algèbre linéaire. Nous nous consacrons par la suite à l'évaluation de performances de notre méthode. Enfin, nous concluons ces travaux dans la dernière partie de ce papier.

2. Moteurs exécutifs

L'utilisation d'environnements de programmation est un moyen efficace pour programmer les architectures parallèles. Certains embarquent leur propre compilateur (OpenMP, OpenACC, ...) avec un moteur exécutif associé, alors que d'autres fournissent uniquement des API du moteur exécutif sous-jacent. Les moteurs exécutifs permettent de faire le lien entre le matériel et l'application afin de programmer ces architectures de manière transparente et efficace.
Nous détaillons dans cette partie différents moteurs exécutifs basés sur des modèles de programmation par tâches avec dépendances de données. Parmi eux, le standard de programmation OpenMP [14], OmpSs [5], X-Kaapi [8] et HARTS [10].

2.1. OpenMP

OpenMP [14] est devenu avec les années un standard de programmation très répandu dans les modèles à mémoire partagée. A l'aide de directives de pré-compilation, il permet d'écrire des applications parallèles sans avoir à réécrire intégralement le code. Un pool de threads est instancié à l'entrée d'une région parallèle, et le travail est ensuite distribué entre les threads. Le modèle par tâche est entré en vigueur depuis la version 3.0 du standard grâce à la directive `#pragma omp task`, qui a ensuite été étendu par l'ajout des dépendances de données dans la version 4.0. L'ordonnancement des tâches prêtes dépend de l'implémentation du compilateur, et il n'existe aucune information à ce sujet.

2.2. OmpSs

OmpSs [5] fournit un modèle de programmation par tâches utilisé aussi bien pour les architectures homogènes (cluster composé de processeurs multi-cœurs) que hétérogènes (multi-GPU). OmpSs est implémenté au dessus du compilateur Mercurium, et du moteur exécutif Nanos++. Il permet d'annoter un code de la même manière que OpenMP, en spécifiant les dépendances sur les données à l'instanciation d'une tâche. Plusieurs politiques d'ordonnancement sont fournies, qui peuvent prendre en compte aussi bien la topologie NUMA de l'architecture que des politiques de priorité des tâches.

2.3. X-Kaapi

X-Kaapi [8] est une bibliothèque C permettant la création non bloquante de tâches. Le parallélisme est explicite alors que la détection de synchronisations est implicite. Les machines visées sont aussi bien homogènes que hétérogènes [9, 12]. X-Kaapi propose plusieurs APIs (C, C++, Fortran) et fournit un moteur exécutif OpenMP avec la compatibilité des binaires sur le moteur exécutif libGOMP de GCC. X-Kaapi est structuré autour de la notion d'ordonnancement par vol de tâches et des ouvriers qui exécutent le code des tâches et prennent des décisions locales d'ordonnancement. Dans ce papier, nous expérimentons une heuristique basée sur l'affinité au sein d'un banc NUMA, qui utilise des piles de tâches par banc NUMA pour placer les tâches là où sont allouées les données.

2.4. HARTS

Le moteur exécutif HARTS [10] repose sur des concepts abstraits. La bibliothèque est basé sur plusieurs modèles. Le modèle d'architecture est décrit via Hwloc [7] pour fournir des informations sur la topologie de l'architecture. Le modèle de tâche permet de décrire une tâche avec plusieurs représentations selon les architectures. Ces tâches sont ensuite gérées via un pool centralisé de tâches. Un algorithme est décrit par un Graphe Acyclique Direct (DAG) avec les dépendances entre tâches. Les DAGs et les tâches sont persistants entre différentes itérations d'un programme. Le modèle de données permet d'encapsuler les données associées aux tâches. Le modèle d'exécution maintient la cohérence entre ces modèles, en distribuant les tâches prêtes sur les ressources disponibles via l'ordonnanceur. Une racine du DAG est donnée à chaque ressource lors de l'exécution.

3. Cas d'étude : les solveurs linéaires

Nous avons développé une API nous permettant de décrire chaque algorithme indépendamment du moteur exécutif utilisé. Dans cette partie, nous présentons notre API et détaillons comment sont partitionnées les données selon la topologie de l'architecture.

3.1. API abstraite d'algèbre linéaire parallèle

Les algorithmes de solveurs linéaires, en particulier les méthodes itératives de Krylov, peuvent être vus comme des séquences successives d'opérations algébriques qui sont exécutées à chaque itération. Ces opérations peuvent aussi bien être des opérations de type BLAS (niveau 1 ou 2), que des produits matrice vecteur creux ou l'application d'un préconditionneur. Nous proposons ici une autre manière d'implémenter des méthodes itératives parallèles en utilisant une programmation par tâche, nous permettant par la suite de viser différents moteurs exécutifs. Notre API abstraite nous permet de cacher aux utilisateurs les spécificités des moteurs exécutifs, comme l'instanciation des tâches, les outils de partitionnement des données et la construction du DAG de tâches. Une des séquences du solveur BiCGStab [15] est décrite dans l'algorithme 1, puis ensuite traduite avec l'API dans le listing 1.
Nous avons implémenté cette API au dessus de différents moteurs exécutifs tels que OpenMP, OmpSs, X-Kaapi et HARTS. La plupart des moteurs exécutifs détruisent les tâches après leur exécution et on les ré-instancie à chaque itération. Parmi ceux présentés, HARTS est le seul à permettre la persistance des tâches : la structure du DAG est construite une seule fois à travers l'API et les tâches sont instanciées à cette construction. Pour les autres, on stocke des structures de tâches propres à l'API (avec les pointeurs sur les données associées au calcul de la tâche) dans un vecteur, i.e. une séquence. A chaque exécution de la séquence, les tâches sont instanciées dans le moteur exécutif visé. Lors de la création de la tâche, certaines annotations peuvent être fournies pour le placement de celles-ci selon une stratégie définie. Certains mo-

Algorithm 1: BiCGStab Algorithm

```
Matrix A;
Vector b, p, pp, r, v;
Scalar a;
do
    pp = inv(P).p;
    v = A.p ;
    r += v;
    a = dot(p,r);
    if(a==0) break;
    ...;
while (|r|<tol*|b|);
```

Listing 1 – BiCGStab sequence

```
AlgebraKernelType alg;
Matrix A; Vector p,pp,r,v;
double alpha;
SequenceType seq = alg.newSequence();
alg.exec(precond,p,pp,seq) ;
alg.mult(A,pp,v,seq);
alg.axpy(1.,r,v,seq);
alg.dot(p,r,alpha,seq);
alg.assertNull(alpha,seq);
while(!iter.stop())
{
  alg.process(seq);
}
```

teurs (comme OmpSs et XKaapi) permettent de placer les tâches en fonction de la localité des données ou simplement sur un nœud de calcul spécifique. Pour XKaapi, une simple clause dans la définition de la tâche suffit. Pour OmpSs, il faut appeler une fonction interne au moteur et spécifier où sera exécuter la tâche.

3.2. Partitionnement et localité des données

Les techniques de parallélisation de solveurs itératifs sont décrites dans [15] et reposent sur le partitionnement du graphe d'adjacence de la matrice, $G_A = (V, E)$. L'ensemble des sommets V représente les lignes ou les colonnes de la matrice, tandis que E désigne ses valeurs non nulles. Les algorithmes de partitionnement permettent de diviser ce graphe en sous-graphes, où chaque partition k est responsable d'un sous-ensemble de G_A noté $G_k = (V_k, E_k)$. Une tâche reçoit alors un pointeur vers la sous-matrice ainsi formée par ce sous-ensemble.

Nous exécutons en parallèle des tâches d'allocation et d'initialisation pour chaque sous-ensemble V_k. De cette manière, les matrices sont allouées sur les bancs NUMA où sont exécutées ces tâches. Cette initialisation parallèle nous permet de pouvoir bénéficier de certaines optimisations dans le cas où le moteur exécutif prend en compte la localité des données au moment d'ordonnancer les tâches. Dans ce cas, on annote la tâche pour qu'elle s'exécute sur le même banc NUMA où a été initialisée la matrice.

4. Étude comparative : de la description des tâches jusqu'à l'exécution

Dans cette partie, nous comparons les comportements des moteurs exécutifs présentés en section 2 sur un solveur linéaire BiCGStab [15] préconditionné. Les systèmes linéaires proviennent soit d'un ensemble de matrices M_R extraites de simulations réelles de réservoir, soit d'un autre ensemble de matrices M_{Lp} venant de la discrétisation aux volumes finis d'un problème de Laplace 2D sur un cube unitaire. Nous notons N le nombre de lignes de la matrice. La machine utilisée pour nos expérience est composée de 2 bancs NUMA reliés par un bus QPI, où chacun possède 32Gb de mémoire et un processeur 8 cœurs Intel Xeon E5-2670 cadencé à 2.60GHz.

Dans un premier temps, nous cherchons à mettre en avant les différences de performances entre les différentes exécutions sur des systèmes aux tailles différentes. Nous cherchons par la suite à extraire les surcoûts engendrés par les moteurs exécutifs. Enfin, nous cherchons à voir les mécanismes en place prenant en compte la topologie des architectures NUMA. Dans les expériences présentées, T_{seq} désigne le temps séquentiel pour exécuter une séquence d'opérations sans moteur exécutif, et T_p le temps pour exécuter celle-ci sur p processeurs.

Nous utilisons ici la version OpenMP 4.0 implémentée par le compilateur Gnu GCC 4.9.0, OmpSs avec les versions 0.10a de Nanos++ et 1.99.9 de Mercurium et enfin X-Kaapi en version 3.1.0 rc10 [1].

1. Version en développement

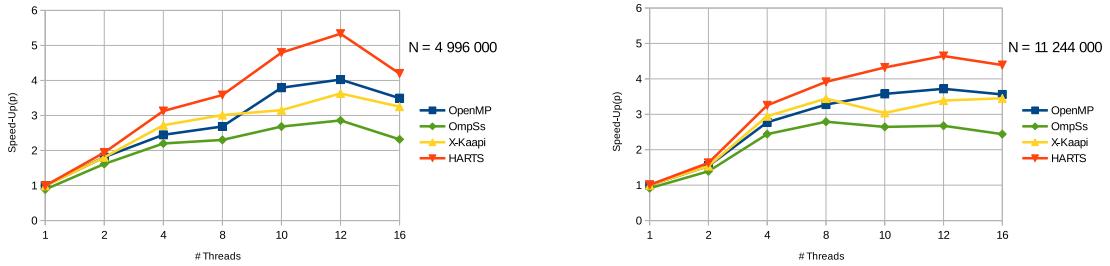

FIGURE 1 – Accélération du BiCGStab sur 2 bancs NUMA

4.1. Comparatif d'efficacité parallèle

Dans le but de comparer les différentes implémentations d'un solveur BiCGStab avec un préconditionneur polynomial [3], nous exécutons successivement cette application sur p threads, parmi $p = \{1, 2, 4, 8, 10, 12, 16\}$, sur la collection de matrices M_{Lp} avec différentes tailles N. Nous calculons ensuite pour chaque exécution l'accélération, $S(p) = T_{seq}/T_p$. Les résultats sont présentés dans la Figure 1.

En analysant les courbes, on peut voir qu'elles ont toutes la même tendance, ce qui est sans doute dû au fait que nous parallélisons cet algorithme sur le même modèle. Lorsqu'on passe sur le second banc NUMA (i.e. $p > 8$), on voit que les écarts augmentent entre les courbes. Sur un seul banc NUMA, l'écart entre les courbes s'explique par les surcoûts liés aux mécanismes de gestion des tâches. En revanche, au delà d'un socket, les mécanismes mis en places par les moteurs exécutifs pour prendre en charge la localité des données à l'ordonnancement sont également impliqués et expliquent des écarts de plus en plus grands : ces écarts sont d'autant plus importants lorsque l'initialisation parallèle du système est inactive.

4.2. Surcoût des moteurs exécutifs

Nous cherchons maintenant à évaluer de manière plus générale les écarts de surcoûts entre les différents moteurs exécutifs. Nous exécutons un solveur BiCGStab avec préconditionneur diagonal sur des matrices issues de M_R de taille N différentes, sur plusieurs threads d'un seul banc NUMA, $p = \{1, 2, 4, 8\}$, avec un nombre fixe d'itérations, $N_{iter} = 1000$. Nous analysons ici l'efficacité parallèle, $Eff(p) = ((T_{seq}/T_p)/p)$, de chaque implémentation. Les résultats sont présentés dans la figure 2, pour une taille de matrice donnée.

Le surcoût d'un runtime par rapport à un autre est représenté par la différence entre les deux courbes d'efficacité. En observant les courbes, on voit qu'elles ont toutes la même tendance en fonction du nombre de threads. Nous avons préalablement vérifié que les performances des différents moteurs exécutifs ne varient pas beaucoup selon la taille du système. OpenMP et X-Kaapi présentent des performances similaires, bien que le second a démontré lors d'une précédente expérience ses faibles taux d'insertion de tâches. HARTS présente des performances plus élevées que les autres car il n'a pas à instancier les tâches à chaque itération contrairement aux autres moteurs exécutifs. Sur un nombre inférieur d'itérations, les coûts d'insertion de tâches dans HARTS sont légèrement supérieurs à X-Kaapi.

4.3. Placement de tâches sur bancs NUMA

L'optimisation des temps de latence passe par l'ordonnanceur qui se doit de prendre en compte la localité des données. Pour évaluer ce facteur, nous avons exécuté un solveur parallèle avec un préconditionneur polynomial sur p threads, équitablement répartis sur N_{Numa} bancs NUMA, où $p = \{8, 24, 48, 96, 144, 192\}$ et $N_{Numa} = \{1, 4, 6, 12, 18, 24\}$. Le système utilisé de taille $N = 62\,480\,000$ est issu de M_{Lp}. Nous comparons les comportements observés pour les implémen-

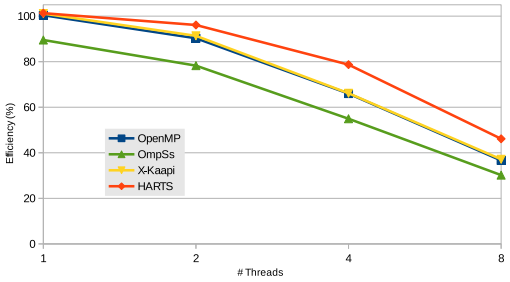

FIGURE 2 – Prec. Diagonal – N = 2 188 842

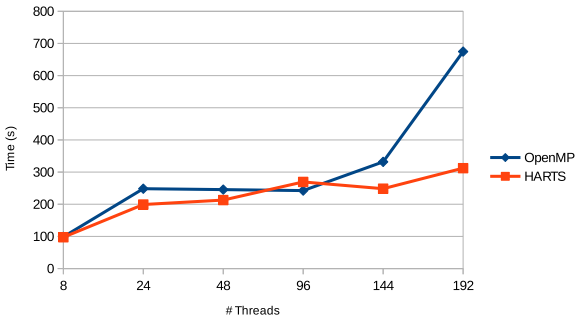

FIGURE 3 – Facteur NUMA et BiCGStab

tations OpenMP et HARTS sur une machine composée de 24 bancs NUMA, chacun ayant une mémoire de 32Gb et un processeur 8 cœurs cadencé à 2,40GHz. L'initialisation parallèle est effective pour que les données soient allouées sur des pages mémoires proches du processeur concerné. Les résultats sont illustrés dans la figure 3.

Nous pouvons premièrement remarquer une perte d'efficacité de notre application sur un grand nombre de threads, comme déjà vu dans la section 4.1, qui peut s'expliquer par de trop lourdes communications et un manque de parallélisme. HARTS perd lentement en efficacité, alors que les performances d'OpenMP stagnent jusqu'à 12 bancs NUMA avant de se dégrader brusquement. HARTS distribue les tâches cycliquement entre les processeurs. Celles avec le même identifiant de partition sont donc toujours exécutées par le même processeur tout en évitant de rapatrier des données distantes. Après vérification, OpenMP ne place pas les tâches selon la localité des données, donc la latence augmente et les performances se dégradent.

Dans le cas de X-Kaapi et OmpSs, chacun permet d'annoter les tâches pour les placer selon la topologie NUMA. X-Kaapi utilise un mot clé à la définition d'une tâche pour gérer le placement d'une tâche selon la localité d'une donnée. OmpSs fait appel à une fonction interne du moteur exécutif pour placer explicitement la tâche sur un banc NUMA. En pratique, nous avons vérifié que ces politiques sont bien respectées, mais nos expérimentations sont toujours en cours.

Vecteurs et matrices distribués

Dans un deuxième temps, nous avons regardé plus en détails la multiplication d'une matrice creuse par un vecteur (SpMV), opération essentielle lors de la résolution de tels systèmes linéaires. Contrairement à l'expérience précédente où seules les matrices sont distribuées, les vecteurs le sont également sur l'ensemble des bancs NUMA disponibles afin de réduire les coûts de latence et la contention mémoire. Chaque vecteur est ainsi découpé selon le partitionnement de graphe de la matrice, et à chaque domaine est associé une portion de vecteur. Nous avons fait tourner nos tests avec le moteur exécutif XKaapi (avec interface OpenMP), mais également OpenMP 4.0 provenant de Gcc 5.2.0 et aussi du compilateur Clang version 3.8.0. Les résultats obtenus sur la même machine que l'expérience précédente sont illustrés Figure 4, sur une matrice provenant de M_{Lp}, en prenant la mesure de 500 itérations de SpMV.

Grâce à sa politique d'ordonnancement des tâches prenant en compte la localité des données, X-Kaapi obtient des performances supérieures aux autres moteurs exécutifs. En revanche pour les autres, malgré que les réductions de temps de calculs en fonction du nombre de cœurs utilisés, les performances sont moindres car les tâches ne sont pas bien placées. Cela se vérifie dès que l'on passe sur plus d'un banc NUMA (i.e. $p > 8$) où là le temps de calcul augmente par rapport au runtime X-Kaapi. En revanche, les tâches s'occupant des mêmes partitions sont toujours exécutées sur les même cœurs. De ce fait, l'exécution de celles-ci bénéficient d'un effet de cache

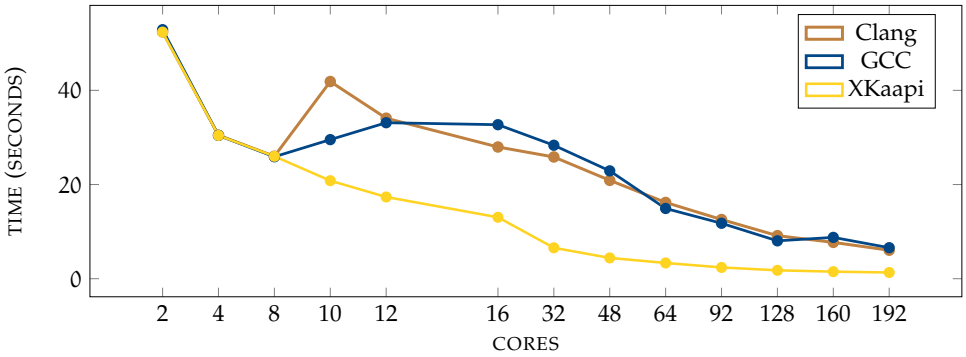

FIGURE 4 – Maille : 2000x2000

permettant de toujours décroitre le temps de calculs. L'exécution d'un graphe de tâches moins équilibré comme pour le BiCGStab aurait pu montrer plus de disparités.

5. Travaux similaires

Bien que des bibliothèques proposent des implémentations parallèles pour les opérations d'algèbre linéaire, aucune ne propose un modèle de programmation par tâches. Nous pouvons notamment citer Hypre [1] qui propose une implémentation à travers un modèle de programmation par passage de message avec MPI, ou MUMPS [2] qui repose sur une implémentation hybride MPI-OpenMP.
Les auteurs de [13] comparent plusieurs moteurs exécutifs OpenMP 3.0 à d'autres bibliothèques ou langages, tels que Cilk et IntelTBB, sur des graphes de tâches déséquilibrés, qui différent des opérations que nous avons évaluées ici. Les résultats maintenant dépassés ont pu montrer d'importantes variations de performances entre les différentes applications.
Dans [11], deux moteurs exécutifs (StarPU [4] et PARSEC [6]) sont proposés pour remplacer l'ordonnanceur spécialisé du solveur linéaire parallèle PaStiX [2]. Bien que les moteurs exécutifs sont efficaces, les auteurs ont pu observer des différences de performances entre les outils.

6. Conclusion

Nous avons implémenté ici un solveur linéaire itératif en utilisant 4 moteurs exécutifs tels que OpenMP 4.0, OmpSs, X-Kaapi et HARTS. Nous avons pu testé chaque implémentation sur des systèmes venant de divers horizons, afin d'étudier le comportement de chacun vis-à-vis de la structure itérative des algorithmes sur des machines parallèles. Nous avons également pu étudier les stratégies adoptées sur des architectures fortement NUMA. Sur un socket, les différences entre les moteurs exécutifs se ressentent dûes aux différentes implémentations des mécanismes internes. Ces différences s'élargissent dès lors que l'on passe sur plusieurs bancs NUMA. Nous avons montré l'intérêt de la persistance de tâches et de DAGs qui sont réutilisés à chaque itération des méthodes itératives telles que les solveurs linéaires. Nous avons également mis en avant l'importance de prendre en compte la localité des données à l'ordonnancement des tâches sur des machines de type NUMA, pour ainsi réduire la latence et gagner en performances. D'autres tests vont être menés notamment sur la gestion de placement de tâches avec X-Kaapi sur les architectures NUMA.

Remerciements :

Ce travail a été partiellement supporté par le projet IRSES2011-295217 HPC-GA. Les auteurs remercient les personnes du BSC pour leurs commentaires et leur aide sur ces travaux.

2. http ://pastix.gforge.inria.fr

Bibliographie

1. Hypre : High performance preconditioners.
2. Amestoy (P. R.), Boiteau (O.), Buttari (A.), Joslin (G.), L'Excellent (J.-Y.), Sid-Lakhdar (W. M.), Weisbecker (C.), Forzan (M.), Pozza (C.), Pellissier (V.) et Perrin (R.). – Shared memory parallelism and low-rank approximation techniques applied to direct solvers in FEM simulation (regular paper). – In *IEEE International Conference on the Computation of Electromagnetic Fields (COMPUMAG)*, Budapest, Hungary, 2013.
3. Anciaux-Sedrakian (A.), Gottschling (P.), Gratien (J.-M.) et Guignon (T.). – Survey on Efficient Linear Solvers for Porous Media Flow Models on Recent Hardware Architectures. *Oil and Gas Science and Technology*, vol. 69, n4, août 2014, pp. pp. 753–766.
4. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – Starpu : A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, vol. 23, n2, février 2011, pp. 187–198.
5. Ayguadé (E.), Badia (R. M.), Bellens (P.), Cabrera (D.), Duran (A.), Ferrer (R.), Gonzàlez (M.), Igual (F.), Jiménez-González (D.), Labarta (J.), Martinell (L.), Martorell (X.), Mayo (R.), Pérez (J. M.), Planas (J.) et Quintana-Ortí (E. S.). – Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, vol. 38, n5, 2010, pp. 440–459.
6. Bosilca (G.), Bouteiller (A.), Danalis (A.), Herault (T.), Lemarinier (P.) et Dongarra (J.). – Dague : A generic distributed dag engine for high performance computing. *Parallel Comput.*, vol. 38, n1-2, janvier 2012, pp. 37–51.
7. Broquedis (F.), Clet-Ortega (J.), Moreaud (S.), Furmento (N.), Goglin (B.), Mercier (G.), Thibault (S.) et Namyst (R.). – hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications. – In IEEE (édité par), *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, février 2010.
8. Gautier (T.), Lementec (F.), Faucher (V.) et Raffin (B.). – X-kaapi : a multi paradigm runtime for multicore architectures. – In *Workshop P2S2 in conjunction of ICPP*, p. 16, Lyon, France, octobre 2013.
9. Gautier (T.), Lima (J. V. F.), Maillard (N.) et Raffin (B.). – Xkaapi : A runtime system for data-flow task programming on heterogeneous architectures. – In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, *IPDPS '13*, IPDPS '13, pp. 1299–1308, Washington, DC, USA, 2013. IEEE Computer Society.
10. Gratien (J.-M.). – An abstract object oriented runtime system for heterogeneous parallel architecture. – In *IPDPS Workshops*, pp. 1203–1212. IEEE, 2013.
11. Lacoste (X.), Faverge (M.), Ramet (P.), Thibault (S.) et Bosilca (G.). – *Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes*. – Research Report nRR-8446, INRIA, janvier 2014.
12. Lima (J. V. F.), Gautier (T.), Danjean (V.), Raffin (B.) et Maillard (N.). – Design and analysis of scheduling strategies for multi-cpu and multi-gpu architectures. *Parallel Computing*, vol. 44, 2015, pp. 37–52.
13. Olivier (S. L.) et Prins (J. F.). – Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, vol. 38, n5, 2010, pp. 341–360.
14. OpenMP Architecture Review Board. – OpenMP application program interface version 4.0, 2013.
15. Saad (Y.). – *Iterative Methods for Sparse Linear Systems*. – Philadelphia, PA, USA, Society for Industrial and Applied Mathematics, 2003, 2nd édition.